

AD-A164 748

SOFT TREE: FAULT TREE TECHNIQUE AS APPLIED TO SOFTWARE
REVISION(U) ARMAMENT DIV (AFSC) EGLIN AFB FL
DIRECTORATE OF SYSTEMS SAFETY J W MCINTEE

1/1

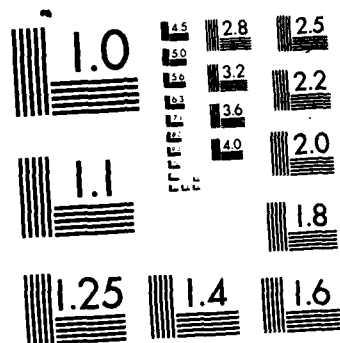
01 OCT 83

F/G 9/2

NL

UNCLASSIFIED

END
4 86



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-164 748

SOFT 3

RAIL - 1
TECHNOLOGY
TO 2011

11

11

11

11

11

11

SOFT TREE

FAULT TREE

TECHNIQUE AS APPLIED

TO SOFTWARE

This document has been approved for "public release."
Questions, comments, or request for copies of this
document may be referred to AD/SES, Eglin AFB, FL
32542.

FOREWORD

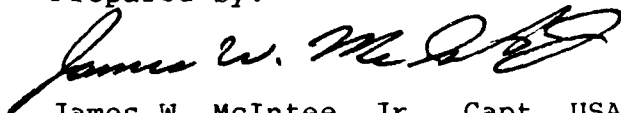
This paper is written with the assumption that the reader understands the basic principles of the Fault Tree Technique, and is geared toward the System Safety Engineer or Manager. The reader should feel free to relay comments, questions, and modifications to:

AD/SES

Eglin Air Force Base FL 32542

or call commercial 904/882-2522, AV 872-2522.

Prepared by:



James W. McIntee, Jr., Capt, USAF
System Safety Program Manager

Reviewed by:



William B. Collins
Director, Systems Safety

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

AD-A164 748

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS None	
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT Public Release/Unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) None		5. MONITORING ORGANIZATION REPORT NUMBER(S) None	
6a. NAME OF PERFORMING ORGANIZATION Armament Division System Safety Engineering		6b. OFFICE SYMBOL (If applicable) AD/SES	
7a. NAME OF MONITORING ORGANIZATION Ballistic Missile Office System Safety Engineering (BMO/AWS)			
6c. ADDRESS (City, State and ZIP Code) AD/SES Eglin AFB, FL 32542-5000		7b. ADDRESS (City, State and ZIP Code) BMO/AWS Norton AFB, CA 92409-6468	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION See 6a		8b. OFFICE SYMBOL (If applicable) See 6b	
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER None			
8c. ADDRESS (City, State and ZIP Code) 6c		10. SOURCE OF FUNDING NOS.	
		PROGRAM ELEMENT NO. PROJECT NO. TASK NO. WORK UNIT NO.	
11. TITLE (Include Security Classification) SOFT TREE Fault Tree Technique as Applied to Software		None None None None	
12. PERSONAL AUTHOR(S) McIntee, James William Jr., Captain (USAF), BMO/AWS			
13a. TYPE OF REPORT Tutorial		13b. TIME COVERED FROM N/A TO N/A	
14. DATE OF REPORT (Yr., Mo., Day) 1983 Oct 01		15. PAGE COUNT 48	
16. SUPPLEMENTARY NOTATION Prepared By: Armament Division Directorate of System Safety Eglin AFB, FL 32542			
17. COSATI CODES (CONT'D REVERSE)		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB. GR.	Soft Tree, Fault Tree, Software, Software Safety, Safety Analysis, Computer Safety, Microprocessor Safety, System Safety, Safety, Weapon Safety, Robotic Safety, Robotics
05	08	-	
09	02	-	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Microprocessors/computers now control many critical functions such as arming and functioning of fuzes. The Software Fault Tree (Soft Tree) begins like any Fault Tree, then continues through the software to the inputs and other electronics. The Soft Tree provides a means of analyzing the hardware/software interfaces and software, as well as, hardware and man/machines interfaces. The technique facilitates the systematic search for safety critical decision points and nodal points in the software. The Soft Tree will also highlight areas where a single bit error can cause a hazardous condition. The Soft Tree is similiar to the Fault Tree in nature; therefore, engineers can use the technique now! <i>James W. McIntee, Jr. System Safety, BMO/AWS</i>			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input type="checkbox"/>		21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Capt James W. McIntee, Jr.		22b. TELEPHONE NUMBER (Include Area Code) (714) 382-4885	
		22c. OFFICE SYMBOL BMO/AWS	

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

17. Con't

<u>Field</u>	<u>Group</u>	<u>Sub Gr.</u>
09	03	-
12	01	-
13	12	-
16	03	-
18	09	-
19	01	-
19	02	-
19	03	-
19	05	-

18. Con't

Non-Nuclear Munition, Non-Nuclear Munition Safety, Fuze Safety, Microprocessor, Automata Safety

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

ABSTRACT

Microprocessors/computers now control many critical functions such as arming and functioning of fuzes. The Software Fault Tree (Soft Tree) begins like any Fault Tree, then continues through the software to the inputs and other electronics. The Soft Tree provides a means of analyzing the hardware/software interfaces and software, as well as, hardware and man/machine interfaces. The technique facilitates the systematic search for safety critical decision points and nodal points in the software. The Soft Tree will also highlight areas where a single bit error can cause a hazardous condition. The Soft Tree is similar to the Fault Tree in nature; therefore, engineers can use the technique now!

TABLE OF CONTENTS

	<u>PAGE</u>
Title Page	i
Foreword	ii
Abstract	iii
Table of Contents	iv
List of Figures	v
Definitions	vi
Acronym List	vii
Acknowledgement	ix
I. Introduction	1
II. Background	1
III. Fault Trees, General	2
IV. The Soft Tree Technique	2
V. Results of Soft Tree Analysis	10
VI. Software Safety Requirements	10
A. Generally Applicable Requirements	10
B. Specific Applications	11
VII. Summary	11
VIII. Applicability	12
IX. Conclusion	12
Appendix A - Example Soft Tree (Fault Tree)	A-1
Appendix B - Detailed Software Description	B-1

LIST OF FIGURES

<u>FIGURE</u>	<u>PAGE</u>
1. Fuze Cross-Section	3
2. Simplified Fuze Electrical Interface	4
3. Example Block Diagram	7
4. Partial Software Flow Diagram	9

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

NOV 1964

DEFINITIONS

Critical - This term describes functions, circuits, activities, and hardware and software components which control, reverse, or apply directly to the authorization, prearm, arm, release, launch, or targeting functions of a weapon system.

Erroneous Bit - A single bit in a register or memory location that was intended to be a "1" which was interpreted as a "0" (or vice versa) during software execution.

Firmware - Software that resides in a nonvolatile medium which is read-only in nature. Firmware is completely write-protected when functioning in its operational mode.

Flow Diagram - A graphic representation of the processing order or execution of instructions and subroutines that make up the software.

Hardware - Physical parts of a system such as mechanical and electrical components, switches and input/output devices.

Microprocessor - The central electronic device which actually executes the software. (Usually surrounded by peripheral devices such as memory, buffers, decoders, etc. which allow interaction with the rest of the system involved.)

Node - A point where several paths meet.

Soft Tree - A term coined to describe a Fault Tree which is constructed on a system which includes software interfacing with hardware. A software Fault Tree.

Software - A series of instructions or statements (including firmware) designed to cause an electronic computer (automation) to execute an operation.

Stray Voltage - An unintended voltage existing in any part of a weapon system.

Volatile Memory - A storage medium that loses information when power is removed from the system.

ACRONYM LIST

CHG - Charge

DET - Detonator

DLY CLK - Delay Clock

FFCS - Fuze Function Control Set

FREQ - Frequency

GP - General Purpose

I/O - Input/Output

IT - Idle till Timer Overflows Instruction

M(,) - Indicates memory location in RAM

M(, ,) - Third number indicates bit within the
referenced memory location in RAM

ms - millisecond

OBD - Instruction to output Bd register to D register

OSD - Office of the Secretary of Defense

PAF - Piston, Arm and Fire

PC - Program Counter

PF - Primary Failure

PTPADT - Prior to Proper ARM Decision Time

PTPGDT - Prior to Proper GAG Decision Time

RAM - Random Access Memory

RET - Return

RETSK - Return Skip

ROM - Read-Only Memory

RTD - Retard

RTDSET - Retard Set

SCR - Silicone Controlled Rectifier

TM - Telemetry

XTAL - Crystal

μ C - Microcomputer

\$ - Indicates number is hexadecimal

ACKNOWLEDGEMENT

A special thanks to Mr. Danny R. Hayles, AD/SES, for his assistance in developing this software Fault Tree concept into the first Soft Tree, and for his detailed knowledge of the microcomputer controlled general purpose bomb fuze system from which the example Soft Tree was constructed.

I. INTRODUCTION. The development of microprocessors/computers has grown at a phenomenal rate in the past few years. Engineers have replaced and simplified complex mechanisms with minimal hardware and microprocessor control. Microprocessors allow for expanded control and added capability with less space and weight. However, they complicate the job of the safety engineer, because there has been no specialized analytical technique for fault analysis of software. Safety engineers can close the technology gap by applying hardware analysis techniques to software analysis. More refined techniques might eventually be developed to analyze software under computer control. This paper addresses an approach to software analysis using the Fault Tree technique. To effectively present this approach to software analysis, it is first necessary to provide a brief background of the problem and some basic Fault Tree philosophy.

II. BACKGROUND

A. The safety community has been concerned with the analysis of microprocessors and their software since they first came into widespread use. Interest in software continued to build as microprocessors began to control more critical functions. In the past, when systems containing microprocessors required analysis, the system was analyzed up to the microprocessor outputs. Some assumptions were then made as to the safety or reliability of the microprocessor itself. In many cases, the microprocessor was allowed to control non-critical functions or merely monitor the hardware that performed the critical functions. Microprocessors have now begun to control more critical functions such as arming and firing of fuzes, weapon release, navigation and control of missiles, aircraft, etc.

B. There have been several papers written concerning software safety. The opinions on failure modes of software and ways to prevent failures and mistakes are numerous and many good points have surfaced that enhance software safety. An important point is that a majority of the software safety problems can be avoided or prevented if safety engineers review requirements documents. This review allows the safety engineer to add safety requirements and real time cross-checks based on past experience with similar systems. This is a good technique to use early in a system's life cycle to head off problems. Other papers establish areas of importance that should be checked by the safety engineer such as the hardware/operator and hardware/software interfaces. More than one paper has suggested the use of a Preliminary Hazard Analysis (PHA), Failure Mode Analysis (FMA), and Failure Modes and Effects Analysis (FMEA) to assist with the establishment of safety requirements. These approaches, all possible and valid, can be augmented later in the system lifecycle by a Software Fault Tree Analysis, which I called Soft Tree Analysis.

III. FAULT TREES, GENERAL. The Fault Tree is an ideal analysis technique where there is a single (or few) undesired event such as premature arming or functioning of a fuze. The Fault Tree technique also works best with systems that involve a "flow" of events. It must also be remembered that a Fault Tree is a model of the system being analyzed. Just as an aeronautical engineer uses a scale model to analyze flow in a wind tunnel, a safety engineer uses the Fault Tree to model a system and analyze the flow of events. As with the wind tunnel models, the more accurately the model represents the system, the more useful the information acquired. It is of great importance that Fault Trees accurately model the system in question if the Fault Tree is used as a basis for design changes. An accurate Fault Tree also allows the tree to be checked for completeness by the customer, management and other engineers.

IV. THE SOFT TREE TECHNIQUE.

A. The term Soft Tree has been coined to describe a Fault Tree which includes software interfacing with hardware. This technique is universally applicable to microprocessor controlled systems where safety is of concern. The example and explanation chosen for this paper have been drawn from a fuze development program.

B. As with any normal Fault Tree, the Soft Tree begins with the top event, for example, "fuze arms prior to safe separation after standard release." In order to proceed further a brief system description is necessary. The example system uses a single rotor to interrupt the explosive train. The fuze detonator is housed in the rotor and is shorted and grounded when the rotor is in the safe position. The rotor is locked in the safe position by the gag rod. The gag rod is removed 100 ms prior to the selected arm time by the piston actuator. When the piston actuator fires, it moves downward and the gag rod is pushed to the right by the leaf spring (figure 1). The rotor is retained by a detent spring until the bellows motor fires. At arm time, the bellows motor fires rotating the rotor to the armed position. Schematically the rotor and electrical interface can be simplified as shown in figure 2. Now we can begin the Soft Tree (Soft Tree, since we know it will include software) (see page A-3). We will investigate removal of the gag rod first, since this must occur before the rotor can rotate. Since the gag rod actually fires 100 ms before arm time, it was necessary to identify this time as the "proper gag decision time." The fault event is therefore "prior to the proper gag decision time" or "PTPGDT." This term is used as we follow the left branch of the soft tree through gates 2, 6. The gag rod lock is removed by the piston actuator. In order to fire the piston actuator (gate 8), the command fault requires the discharge of the PAF

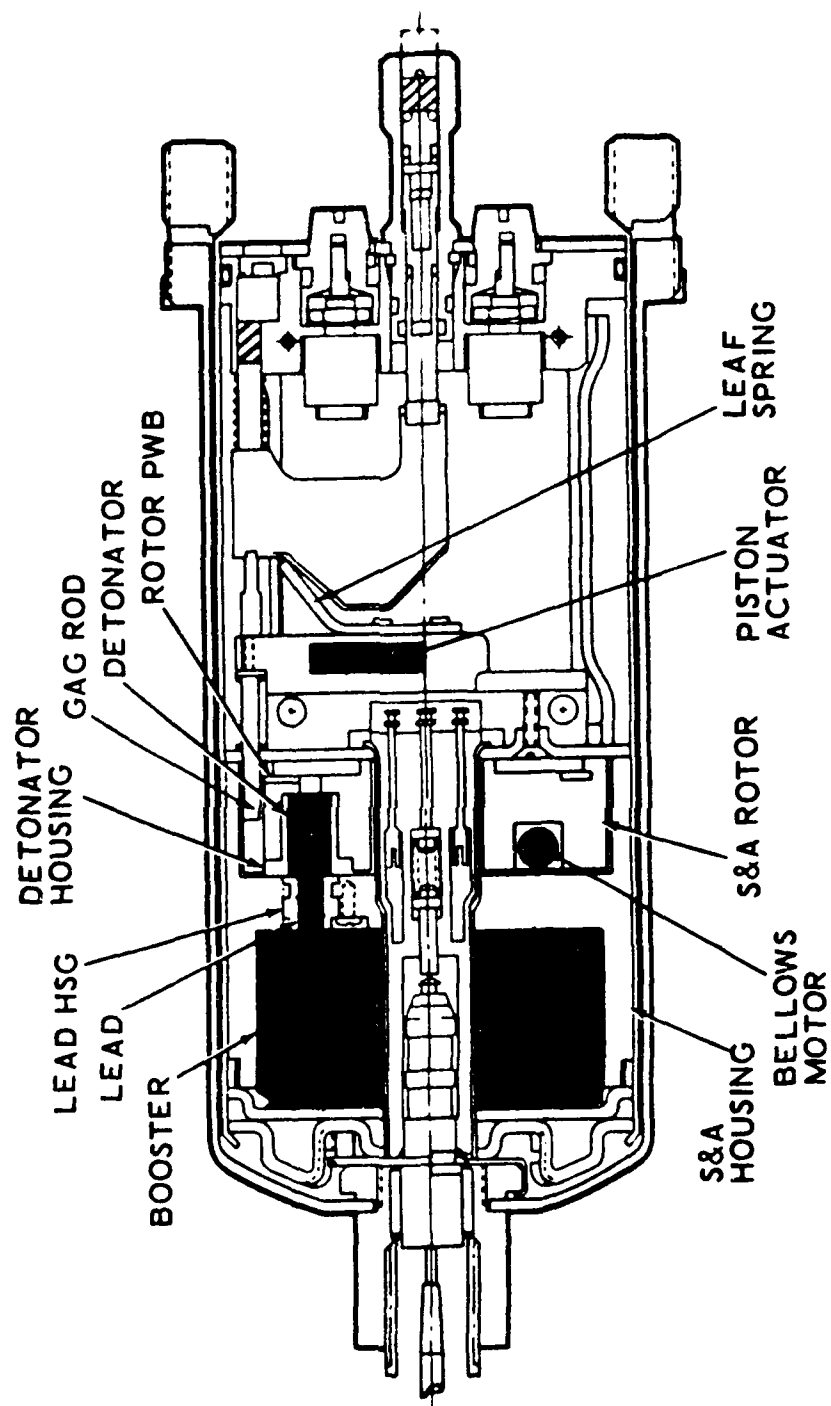


FIGURE 1. FUZE CROSS-SECTION

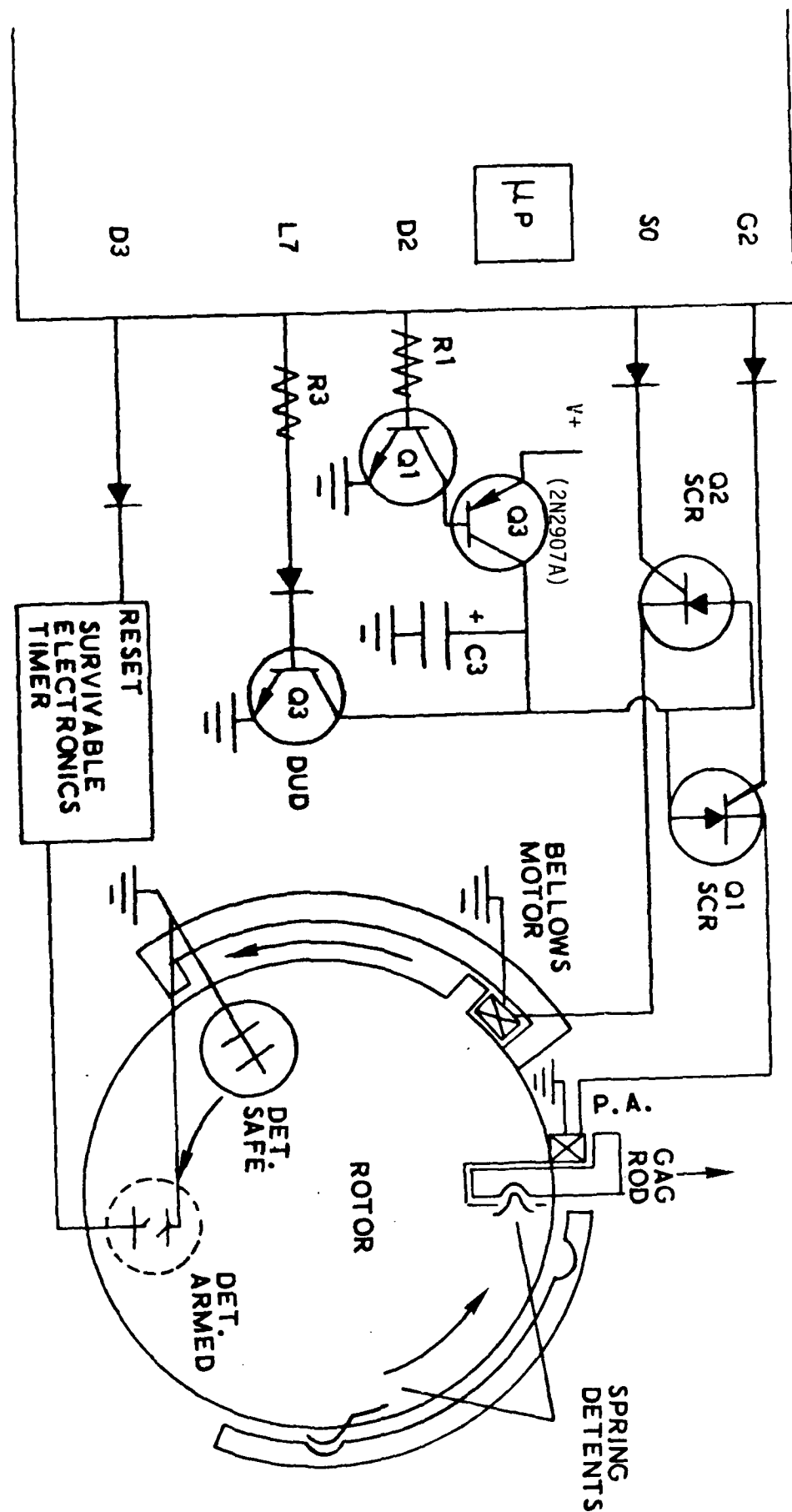


FIGURE 2. SIMPLIFIED FUZE ELECTRICAL INTERFACE

capacitor (gate 9). The state of the system event at gate 9 requires the PAF capacitor C3 be charged AND Q1 SCR conduct. First, we will investigate the charging of the capacitor (transfer I, page A-4). We need power to AND through transistor Q3 (2N2907A) (gate 11). In order for Q3 to conduct, Q1 must conduct to ground. Finally, in order for this transistor to be commanded to conduct to ground, output port D2 must go high.

C. At this point the "normal" Fault Tree would stop and consider a failure of the microprocessor as anything that can cause a high output at D2. However, since the safety of the system depends/relies on the software, the adequacy of this "safety device" must be verified. This is the hardware/software interface. In order to have a complete system Soft Tree, we must cross this interface and include the software in our safety analysis.

D. As a general rule, there are two conditions which must be met in order for any microprocessor to output data. There has to be (1) a binary word (data) available to the output port and (2) an instruction to output the word. This AND gate begins the software analysis section of the Soft Tree. From this point on we must look to software to find out what the minimum immediate necessary and sufficient conditions are to get (1) the binary word AND, (2) the instruction that causes the output. For a 4-bit microprocessor, the binary word that is needed could be anything from 0000 to 1111 depending on the implementation in the system. The binary word could come from various places including the accumulator, the address registers or direct from memory.

E. The instruction must occur "given that the binary word is available for output." The instruction can be either inadvertent or commanded by the microprocessor. Normally the output port is used for several different functions. It is, therefore, important to examine the various locations (in the software) that this output instruction occurs. As each is found, it is necessary to use some engineering judgment as to whether the instruction can be arrived at with the correct binary word available for output. For example, let's assume the word "1000" is required for a specific event "A" to occur; and the instruction "Load Accumulator Out" transfers the word to an output register. Normally "1000" would be a unique word that is only used to cause event "A". However, depending on the microprocessor interfaces, the minimum, immediate necessary and sufficient condition for "A" may be that a "1" is present in the most significant bit location. This would imply that 1000, 1111, or 1XXX could cause event "A" if the output instruction is reached. It is, of course, imperative

that the person(s) constructing the Soft Tree be very familiar with implementation of microprocessors or at least have easy access to an engineer who is familiar with microprocessors and the system in question.

F. In our system, D0, D1, D2, and D3 are the four parallel output lines that tie to the D register, therefore, when we want D2 high, the software should reflect a "1" output to D2. To complete the Soft Tree, we need to "look" inside the microprocessor and find out where the "1" comes from. We use the block diagram to "look" inside the microprocessor (figure 3). The D register is pictured in the center, at the right of the figure. From the diagram (and data book) we find that all outputs from the D register must be transferred from the BD register to the D register. The other requirement is, of course, the availability of the "1" (gate 15). The transfer command for this particular microprocessor was the OBD instruction which loads the contents of BD into D (gate 16). At gate 18, we see the OBD instruction occurs seven times in the software. Five of the seven outputs can only occur if there is an erroneous bit. An erroneous bit is defined as one (or more) bit(s) of a register or memory location that was intended to be a "1" which was interpreted as a "0" (or vice versa) during software execution. An erroneous bit is a hardware failure which manifests itself as incorrect or unpredictable software execution. It is assumed that this is the only type of failure that the microprocessor hardware can be responsible for and that the software program does not "fail". The erroneous bit can, however, take on many forms.

G. An erroneous bit in the program counter causes an inadvertent jump in the execution of the software. An erroneous bit in the scratch pad memory address register causes the wrong memory data location to be accessed. An erroneous bit in memory causes a change in the data/instruction stored in memory. At this point the problem of finding all the combinations of erroneous bits seems monumental. In fact, it is not. The Fault Tree technique considers only the failures in hardware systems which can lead to the undesired event; the same holds true for Soft Trees. If an erroneous bit can cause the specific fault event in question, it is included in the tree. After considering erroneous bit failures including inadvertent jumps to the address with the critical instruction, it is necessary to consult the software flow diagram. Anywhere that software paths come together at a node, an OR gate exists and is drawn into the soft tree. Decision points in the software flow diagram are treated like switches - AND gates. You need flow to the address location AND satisfaction of the decision parameters. Cascades of instructions that do not affect the fault event directly can be lumped together as "flow through address XXXX to address XXXX." If having an event occur prematurely is a fault event, then the control of the clock frequency must be considered. (Some microprocessors

have dual clocks that must be cross-checked). Now that we know the instruction that causes the output we can go to the software and continue the Soft Tree. Figure 4 shows a small portion of the software flow diagram. This is a flow diagram written from the actual machine language program NOT the flow diagram used to write the program (for added detail see Appendix B).

H. In order to ensure that the flow diagram correctly represents the software, it must be very detailed (nearly to the individual instruction) and the flow diagram should be made from the software. Even though a flow diagram is made before the software is written, it will not be detailed enough and may not match the final software. It is imperative that the flow diagram match the software exactly or the Soft Tree model will lose its value. Once done it can be analyzed like electronics using the same computer programs used on Fault Trees. To continue with the Soft Tree, we find that the OBD instruction that charges the PAF capacitor is in the PAF CHG subroutine. The PAF/CHG subroutine, used for charging the PAF capacitor for ungag, is referenced at address \$2C8 which corresponds to the event at transfer 11, page A-5. (Note: \$ - Indicates number in hexadecimal.) If you recall, the fault events have been that the fuze arms early or PTPGDT; therefore, we must investigate how the PAF capacitor could get charged prior to the proper gag decision time. As you can see below gate 19, this program could get to \$2C8 by direct jump or it could get to \$2C8 by entering from the program step above \$2C8. At this point, we need to break away from the software instruction itself and look at what can cause the software to execute instructions in the correct sequence but too fast. The software could arrive at \$2C8 early by fast program execution due to a fast clock, as shown under gate 20, or the program could arrive at \$2C8 early with a correct clock but due to some error or fault earlier in the program. Again we go to the software flow diagram (figure 4) to see what the previous steps in the program were. We now work with the software flow diagram to trace the software flow in much the same way as an engineer would use an electrical schematic to trace the electron flow. Moving back into the software one step (instruction or subroutine) at a time we come to a software node where three branches of the software come together above \$2C8. This is an OR gate. The program could arrive at \$2C8 by way of the 2.0 second retard verification decision, (gate 23), the 2.6 second RTD verification (transfer E) OR the wait subroutine after the 4.0 second RTD verification decision. We will trace the 2.0 second branch (gate 23). The next type of software construction we come to is the decision. Remember the software node (like the electrical node) is an OR gate; the decision in software is like the electrical switch (or transistor, valve, solenoid contacts, etc.). The decision, therefore, is handled with an AND gate and like a switch requires flow to it and a decision parameter satisfied, as shown at gate 23. Gate 23 requires that the program arrive at

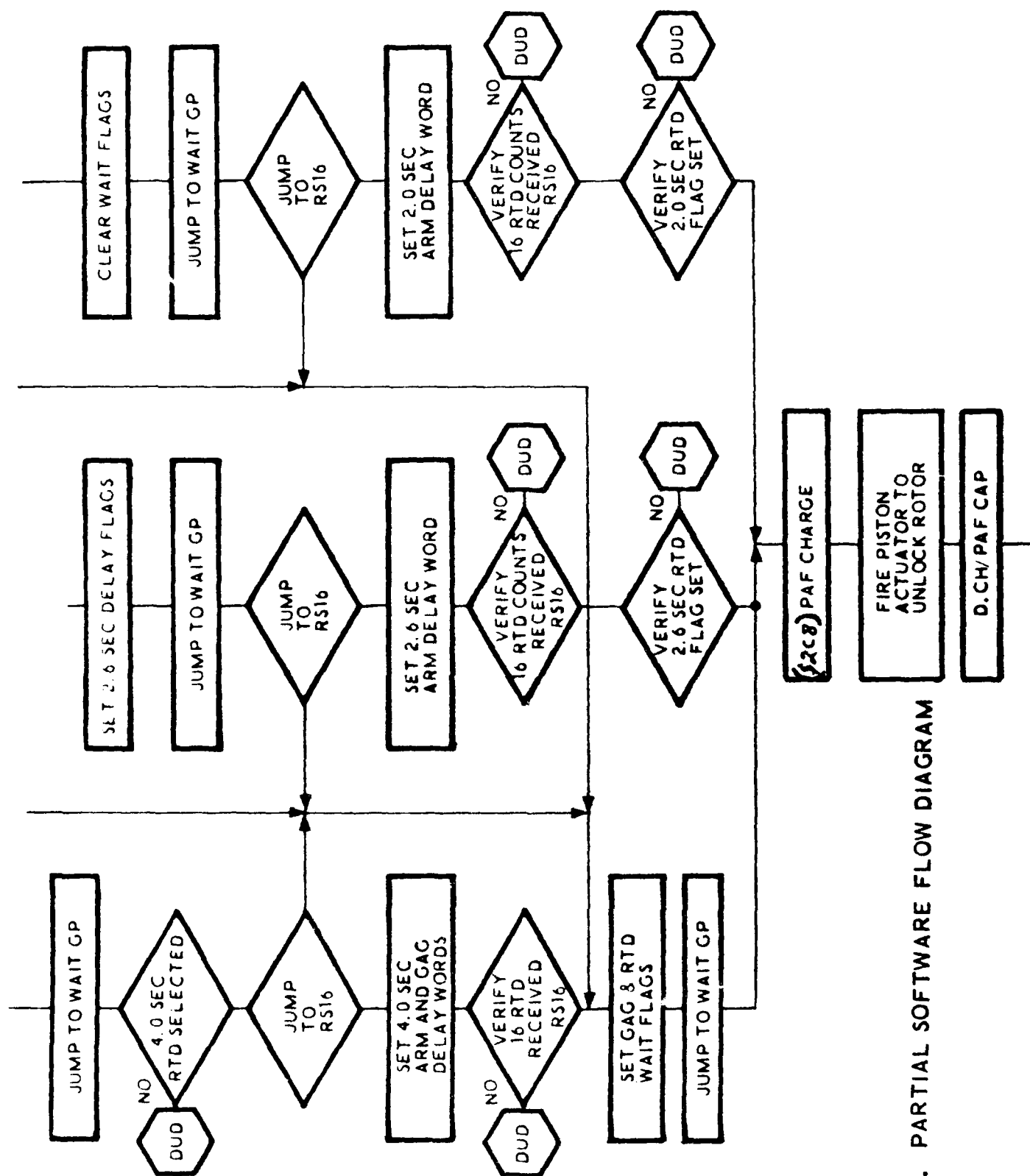


FIGURE 4. PARTIAL SOFTWARE FLOW DIAGRAM

address \$262 and the proper flag set. Gates 24 and 26 follow the flow required to read the delay switches and set the flag. Gate 27 brings the Soft Tree back out of the software to the input lines, L4 -L7, which interface with the electrical delay switches on the front of the fuze. The Soft Tree now can show (by the double diamond) the interface of the electronics with the operator gate 28. This completes a branch of the flow from the undesired event through electronics; to the hardware/software interface; into the software; back through the hardware/software interface, and through the electronics to the hardware/operator interface. In this way, the Soft Tree approach to system safety analysis can truly analyze the entire system for single component and single bit errors that can result in undesired top events.

V. RESULTS OF SOFT TREE ANALYSIS. Upon analysis of this software system, it was noted that there are relatively few steps required to set the critical ARM flags. Furthermore, the flags were set prior to the decoding of the ARM/DELAY switches and not used until just before fuze arming. Since the flags were set, this would allow an inadvertent jump to skip all of the ARM delay decode software and still function the fuze early depending on the ARM delay word in RAM. The setting of the four-byte flag is very critical to the safety of the fuze. Therefore, it was decided that two-bytes of the ARM flag should be set prior to decoding the ARM/DELAY switches and the other two-bytes should be set after the ARM delay word is stored in RAM. If an inadvertent jump occurred and the ARM delay word was not decoded and stored in RAM, all of the ARM flags would not be set. The microcomputer would detect this incorrect program (software) flow and would enter into the "DUD" subroutine.

VI. SOFTWARE SAFETY REQUIREMENTS. The list below is provided as a starting point which safety engineers can delete from or add to depending upon the software/computer hardware implementation of the particular system they are working on. This list is based upon experience and lessons learned as I have worked with computer controlled systems.

A. GENERALLY APPLICABLE REQUIREMENTS.

1. The contractor shall identify safety critical software (code, subroutines or modules).
2. The software shall be developed such that the safety critical software decisions are as close as possible to the output they protect.
3. The software shall be developed such that inadvertent jumps are detected and protected against (by restarts, dud, reinitialized subroutines, etc.).

4. The software shall verify safety critical parameters, or variables before an output is allowed. Parity checks or other checks, require two decisions before output. This is similar to the two fault tolerance and places "AND" gates in the Soft Tree.

B. SPECIFIC APPLICATIONS.

1. If a program is very large:

a. The Soft Tree shall be accomplished based on the higher order language. The compiler accuracy must be verified. Carefully scrutinize "optimization compilers".

b. The software (if possible) should be developed such that the majority of the safety critical decisions and algorithms are within a single (or few) software development modules (to facilitate analysis).

2. If timing is critical: (such as general purpose fuzes). The computer's oscillator shall be checked against an independent time base. The reaction of the software to a difference between the clocks will depend upon the type of system being utilized.

3. If a "watch dog" type circuit is implemented, the circuit shall be totally independent of the computer that it monitors and should begin to monitor the computer as soon as possible after computer power-up (i.e., the "watch dog" should not be initialized by the computer).

4. If it is critical that the computer remain on at all times, (i.e., for destruct modes, shut down sequences, etc.):

a. The computer shall be protected against power interrupts, power surges, stray voltages, and gradual depletion of power supplies.

b. Consideration should be given to connectors and sockets to insure continuous continuity, especially in high vibration environments.

VII. SUMMARY.

A. A Soft Tree is a normal Fault Tree that has a section constructed from the software. In order to construct a Fault Tree on software, it is necessary to use a detailed flow diagram made from the final software program. It is also necessary to refer to the microprocessor data book for the architecture and the instruction set for the processor being used. The Soft Tree like the Fault Tree may be reduced in scope to get a less detailed analysis for very large systems.

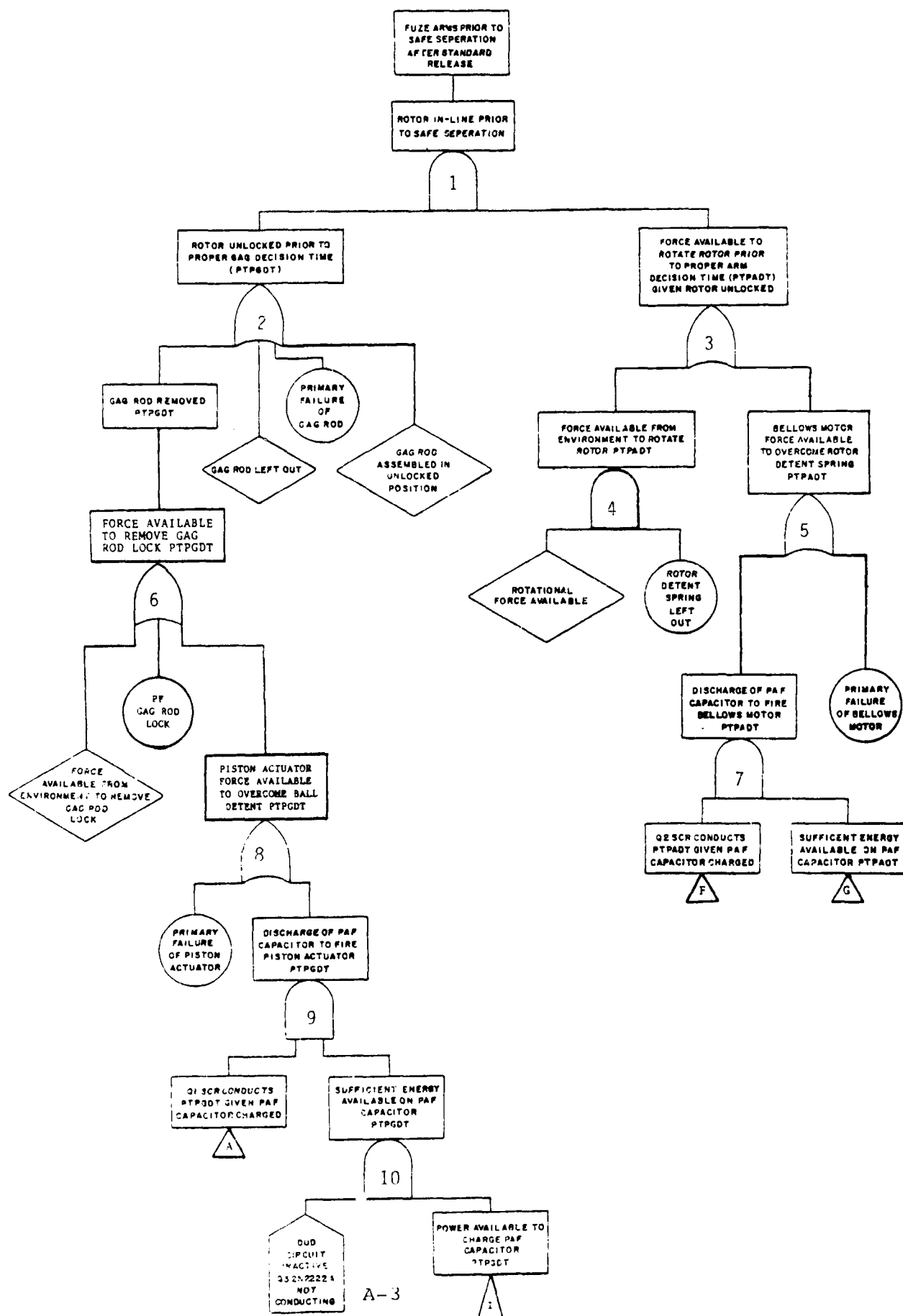
The simplest way to reduce the scope of the Soft Tree on a large system would be to use the higher order language as the basis for the analysis. In this case, special consideration should be given to verify the accuracy of the compiler and other development "tools".

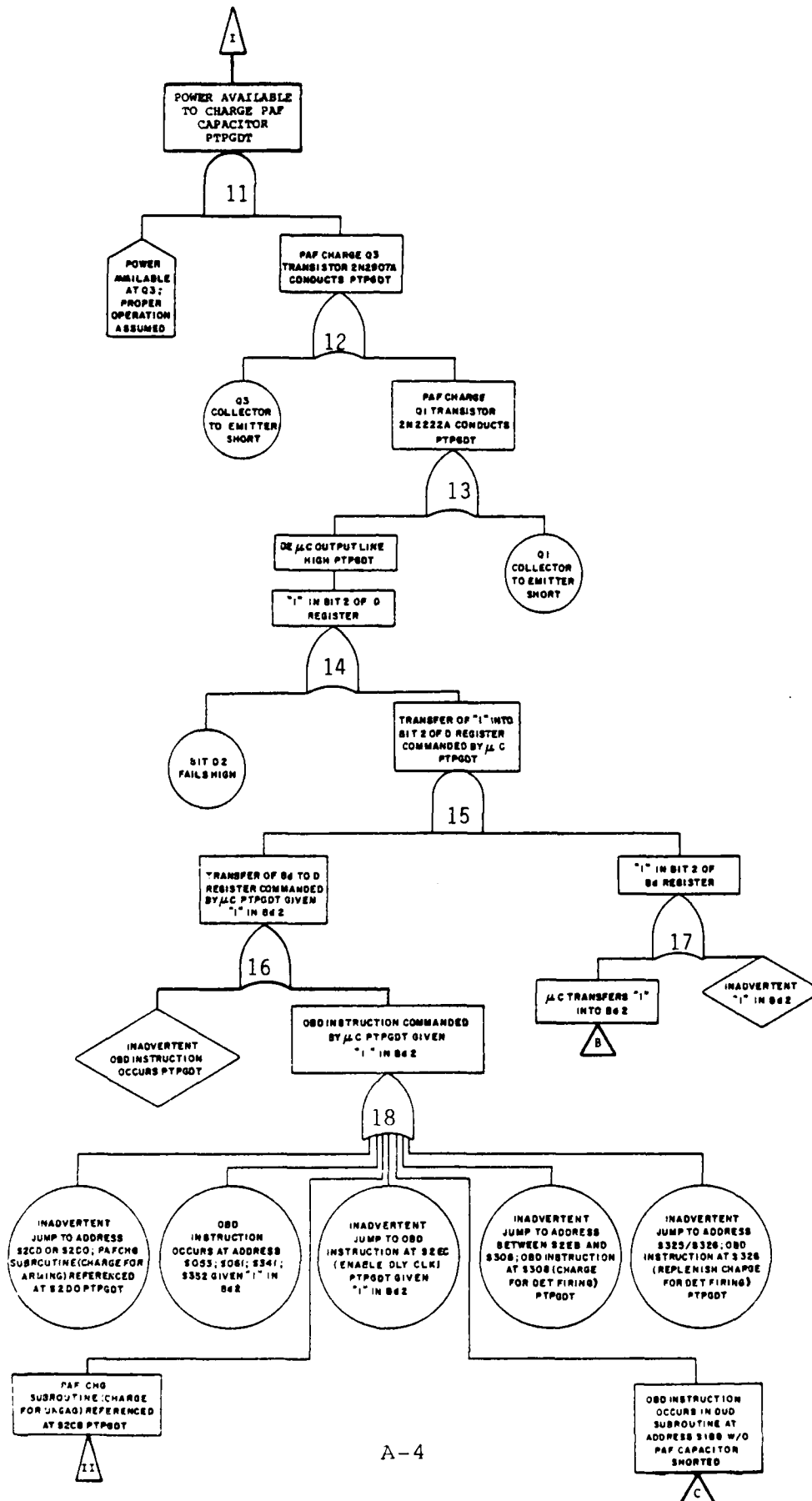
B. Decision points are AND gates, and nodal points are OR gates. We assume the only primary failure that the microprocessor can be responsible for is the erroneous bit (which can occur in the program counter, registers or memory). The final resource that a safety engineer needs to do a successful Soft Tree is a person knowledgeable in microprocessor applications. As with any Fault Tree, the safety engineer must know the system he has to analyze and make every attempt to model the system accurately. The safety engineer must be careful not to jump ahead; but rather work "backwards" through the system (including software) step-by-step, component-by-component, instruction-by-instruction from the top event back through the system.

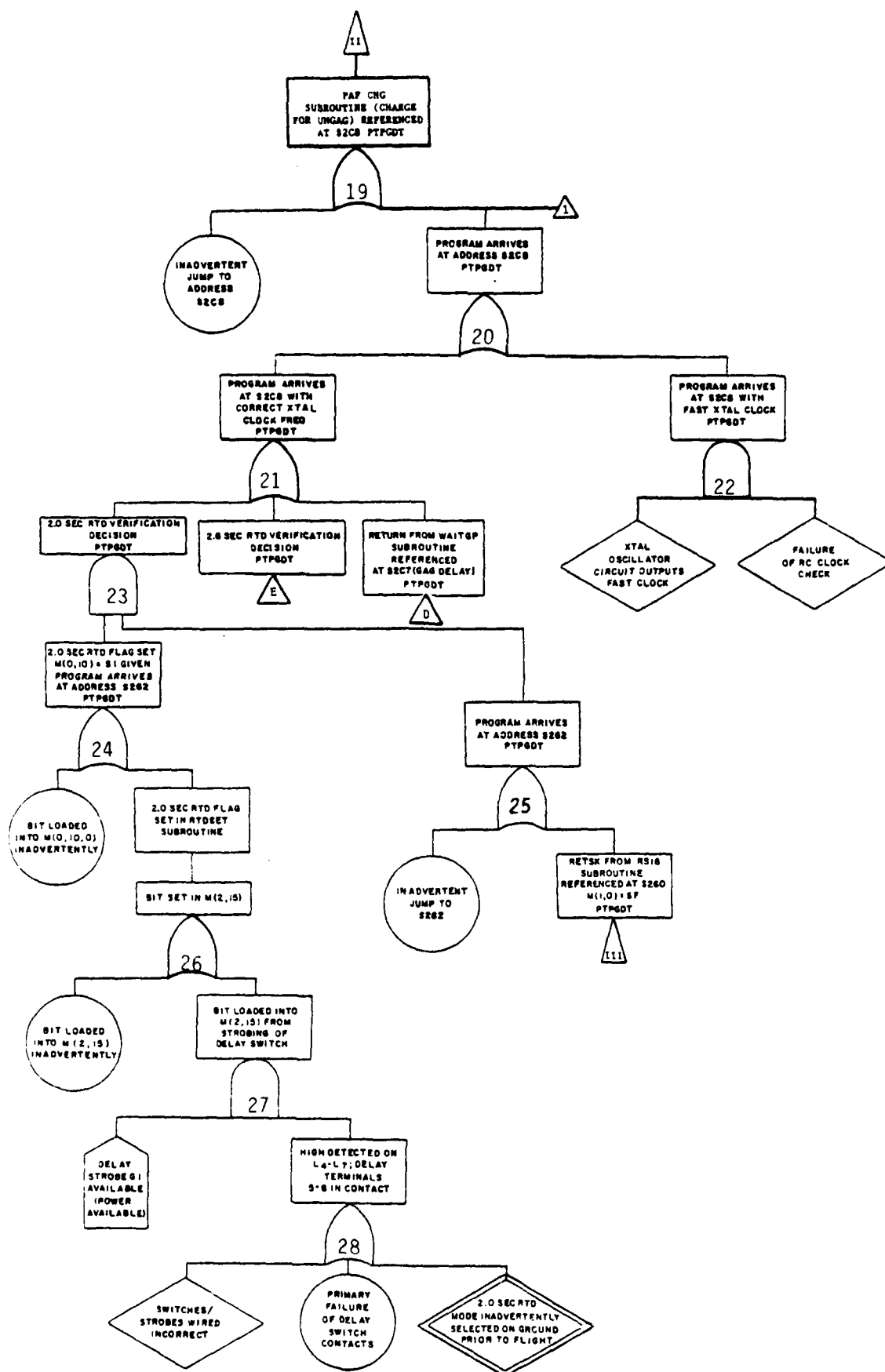
VIII. APPLICABILITY. The Soft Tree can be used on any system that incorporates a microprocessor or microcomputer that interfaces or controls hardware or electronics. The Soft Tree, like the Fault Tree, is best utilized where few very critical top events are of concern and are the result of a "flow" of events.

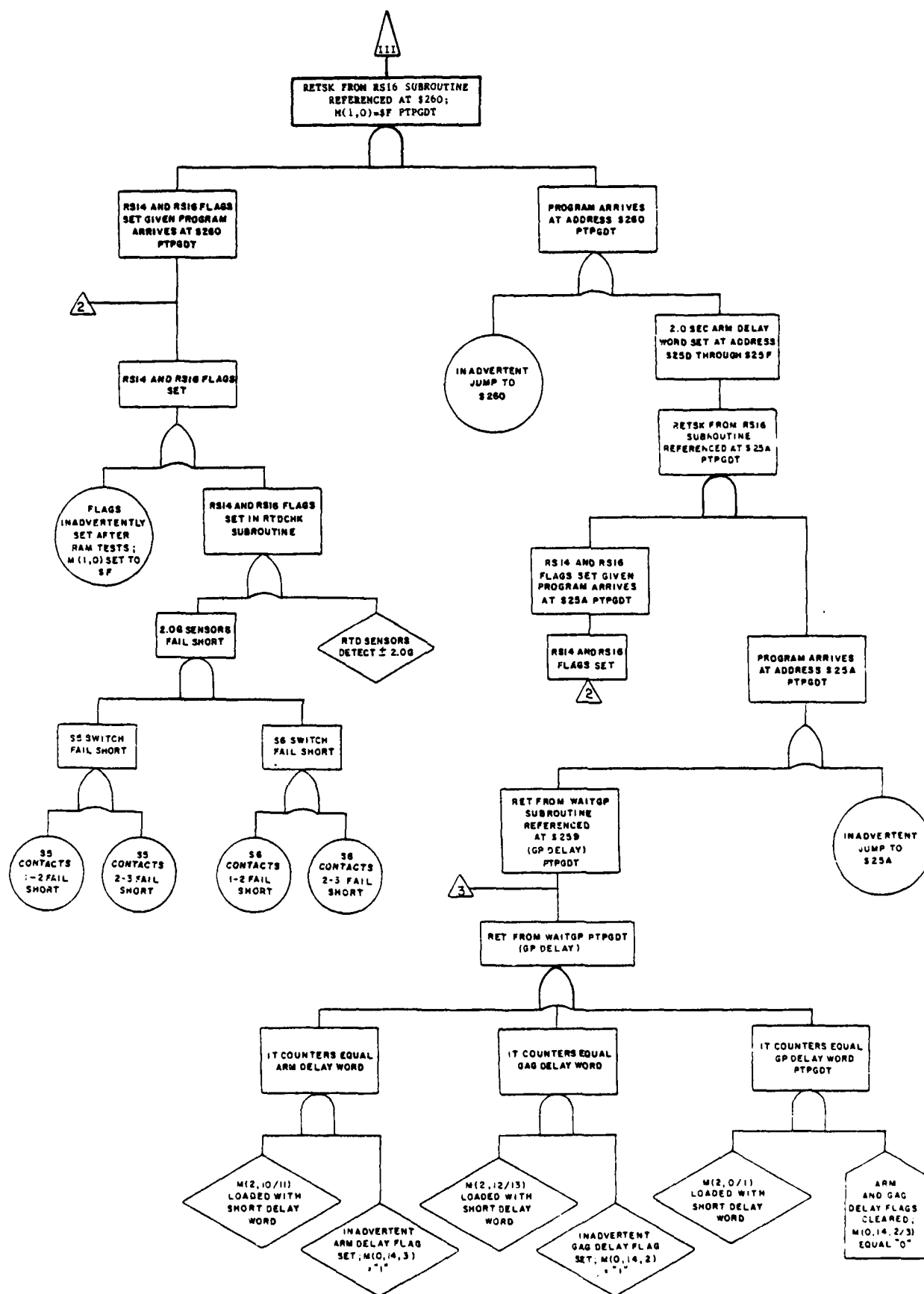
IX. CONCLUSION. The software Fault Tree is a useful technique for finding single and combinations of component hardware failures and single software decisions or instructions that can cause undesired top events. The Soft Tree is enough like the normal Fault Tree that safety engineers, in conjunction with electrical (software) engineers, can use the analysis technique immediately with little trouble. Since every microprocessor controlled system is different, each safety/software engineering team must develop a feel for the extent of detail that should be included in the Soft Tree. The appendix shows excerpts from the Soft Tree developed for a Microprocessor Controlled General Purpose Bomb Fuze.

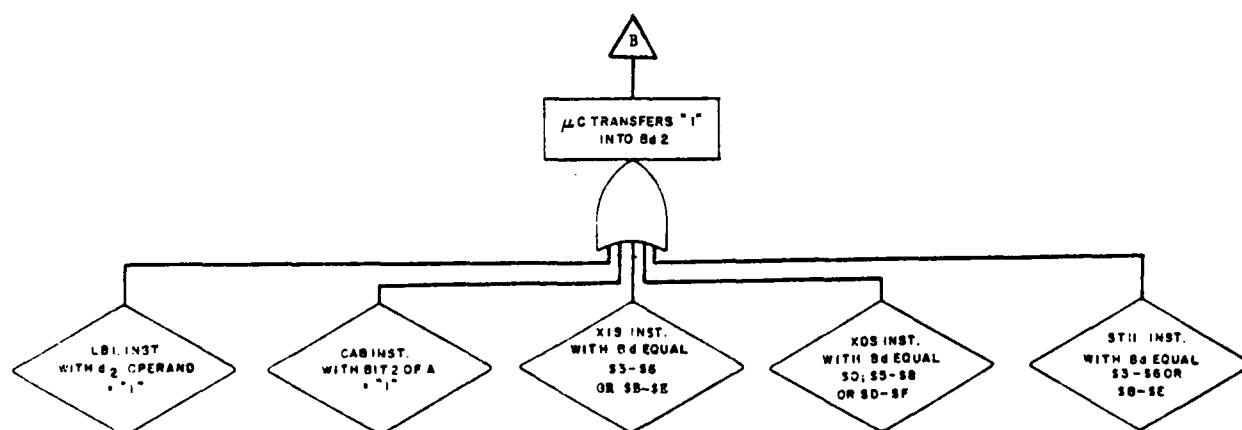
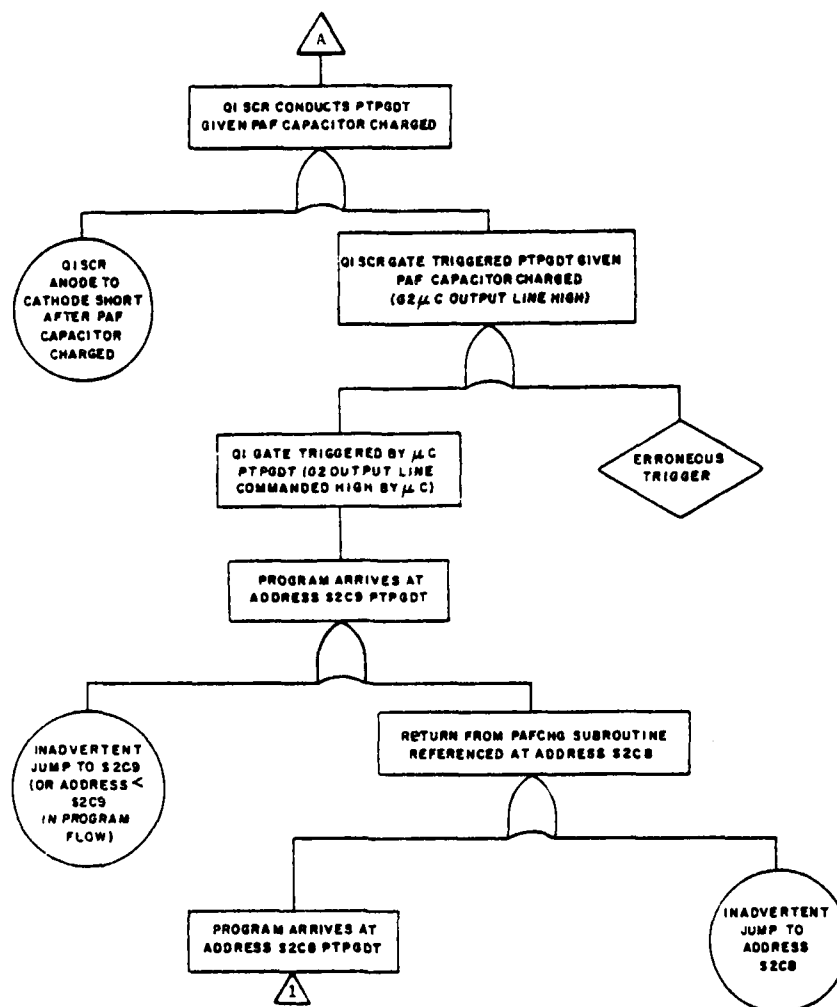
APPENDIX A
EXAMPLE SOFT TREE

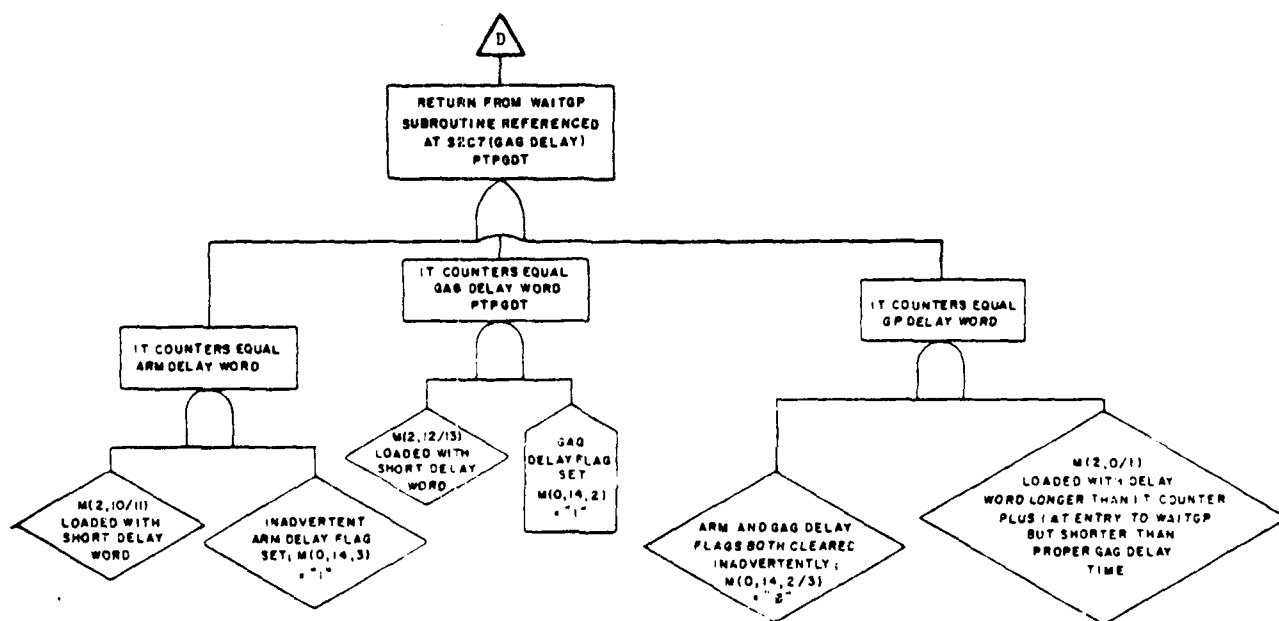
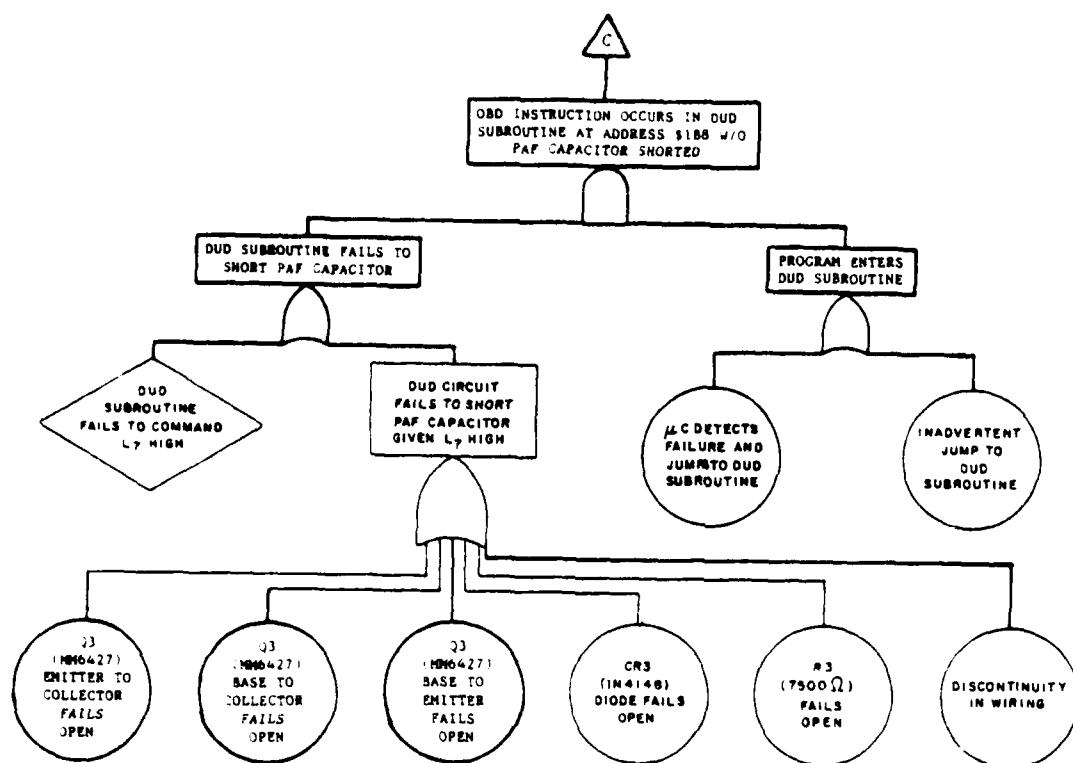


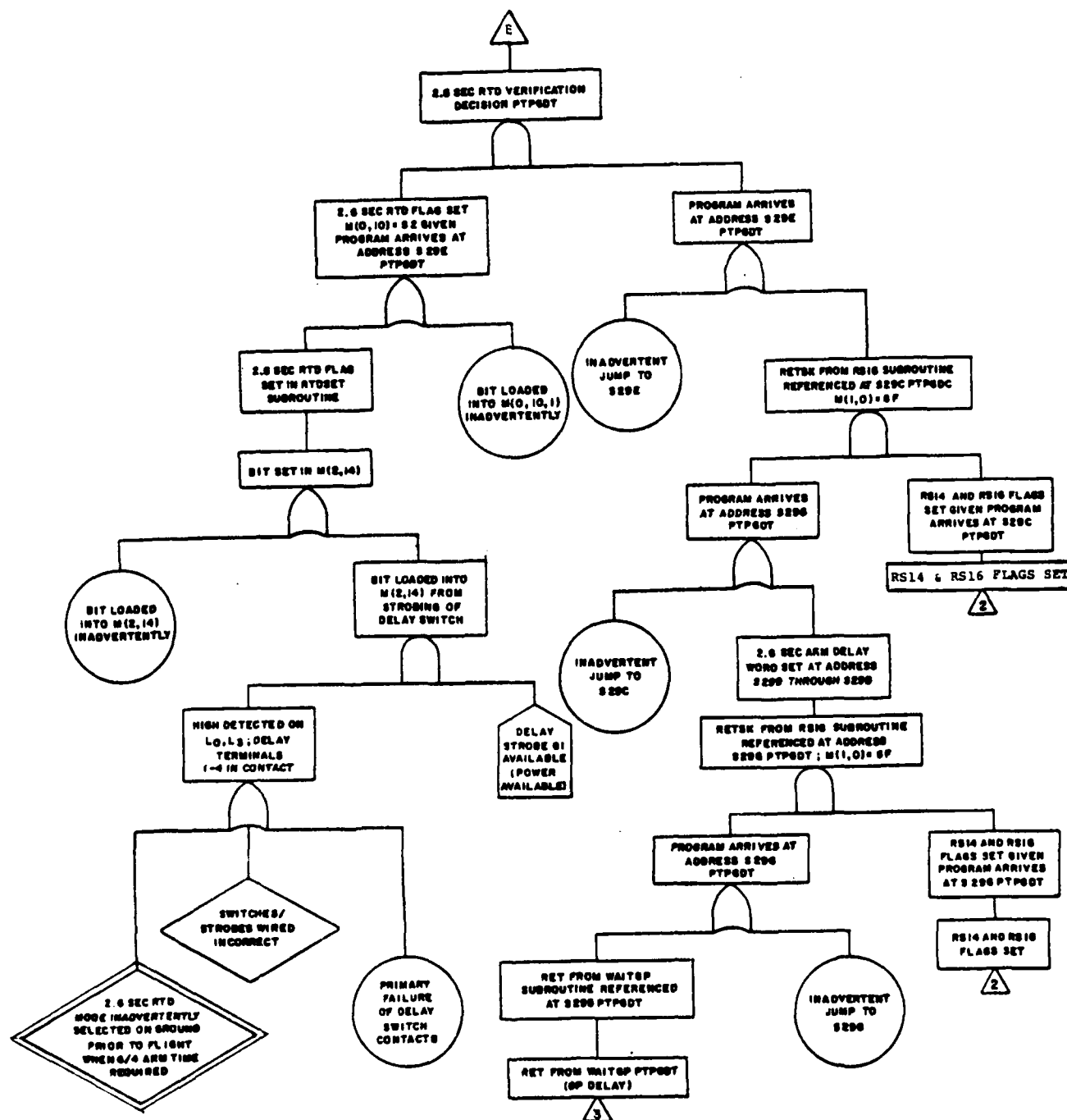


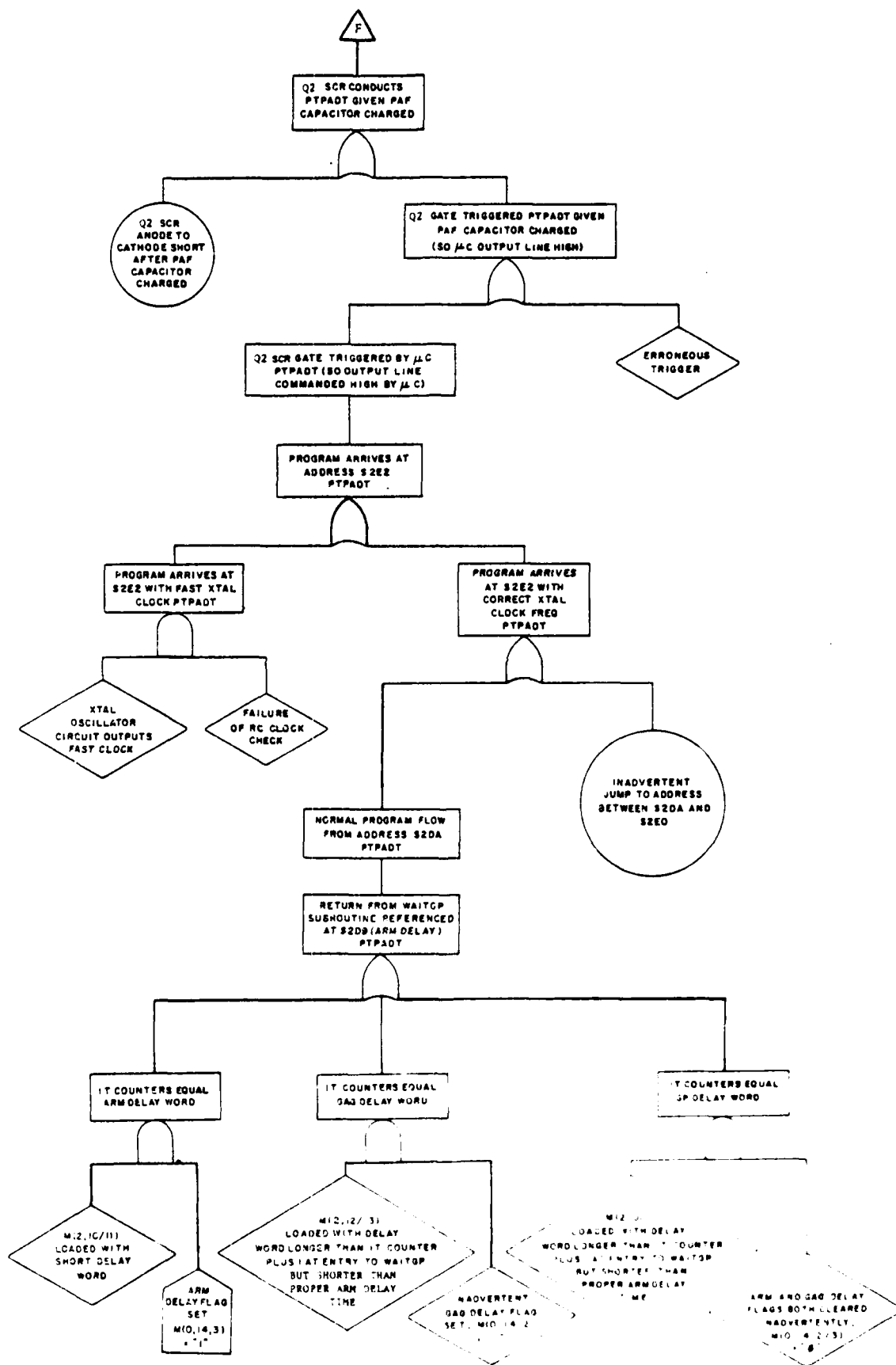


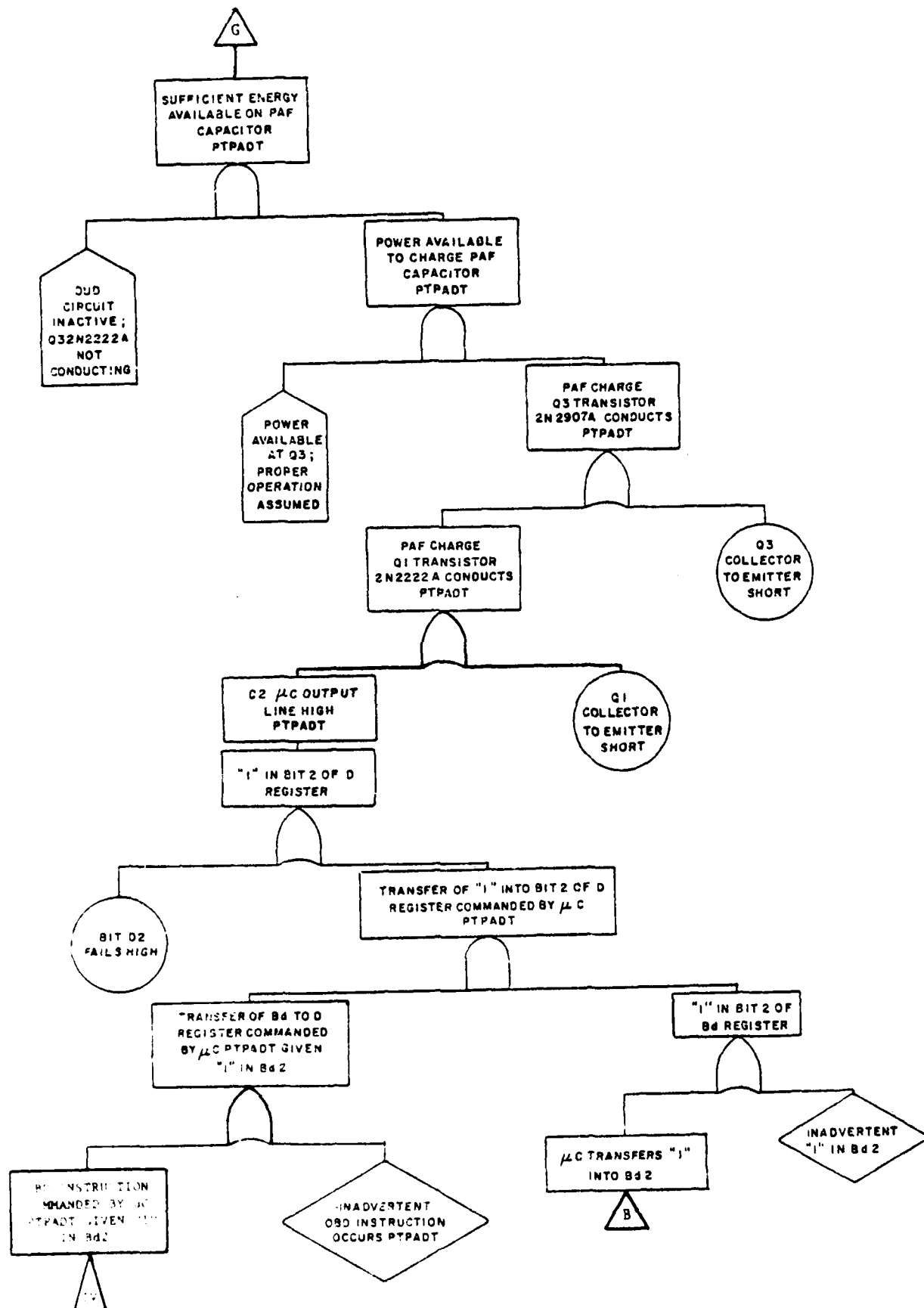


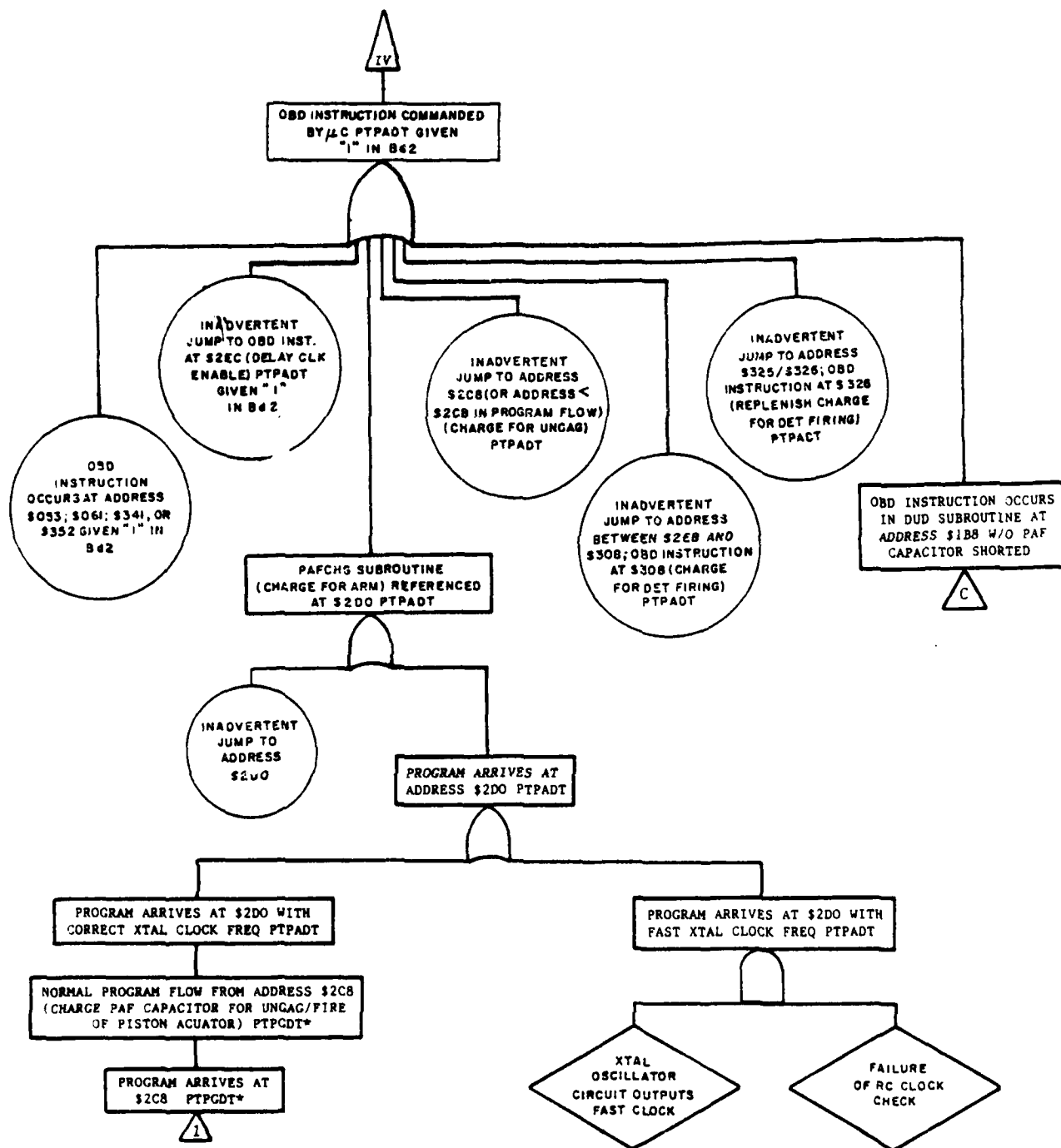












* THIS CHANGE IN REFERENCE REFLECTS THE NORMAL PROGRAM FLOW

APPENDIX B

DETAILED SOFTWARE DESCRIPTION

In the example chosen, the Electronic Bomb Fuze is controlled by a microcomputer. A description of the software for the fuze may be broken down into two areas. The first section provides an overview of the microcomputer architecture and the second section provides a description of the software flow diagram.

1. Microcomputer Architecture. A block diagram of the microcomputer is shown in Figure C-1. It illustrates the inter-connection of the significant blocks within the device.

a. The program memory consists of a 1024-byte read-only memory (ROM). ROM words may be program instructions, program data or ROM address pointers. ROM addressing is accomplished by the 10-bit program counter (PC). Its binary value selects one of the 1024 8-bit words contained in ROM. During program execution, the value of this counter is automatically incremented by one prior to the execution of the current instruction. If the current instruction is a transfer of control, the PC gets loaded from ROM with the address of the next executable instruction. Registers SA, SB and SC provide address storage for a three-level subroutine stack.

b. The data memory consists of a 256-bit random access memory (RAM), organized as four data registers of 16 4-bit digits. RAM addressing is implemented by the 6-bit B-register whose upper 2-bits (Br) select one of four data registers and lower 4-bits (Bd) select one of 16 4-bit digits in the selected data register. The convention for defining any specific RAM bit is given by $M(r,d,x)$, where r identifies the value of Br, d identifies the value of Bd, and x identifies the specific bit of the 4-bit data location pointed to by $M(r,d)$.

c. Following is a brief description of the remaining registers.

(1) The 4-bit ACC (accumulator) register is the source and destination for most I/O (input and output), logic and data memory access operations.

(2) The ALU (arithmetic and logic unit) performs the arithmetic and logic operations.

(3) IN_0 - IN_3 are four general purpose high impedance input ports which may be loaded directly to the ACC-register.

(4) The D-register provides four general purpose output ports which are controlled by setting the appropriate bits of the Bd-register and executing the OBD instruction.

(5) The G-register contents are output to the four general purpose bidirectional I/O ports. Data on the G-ports may be read directly to the ACC-register or transfer of control may be executed depending on the state of the G-ports.

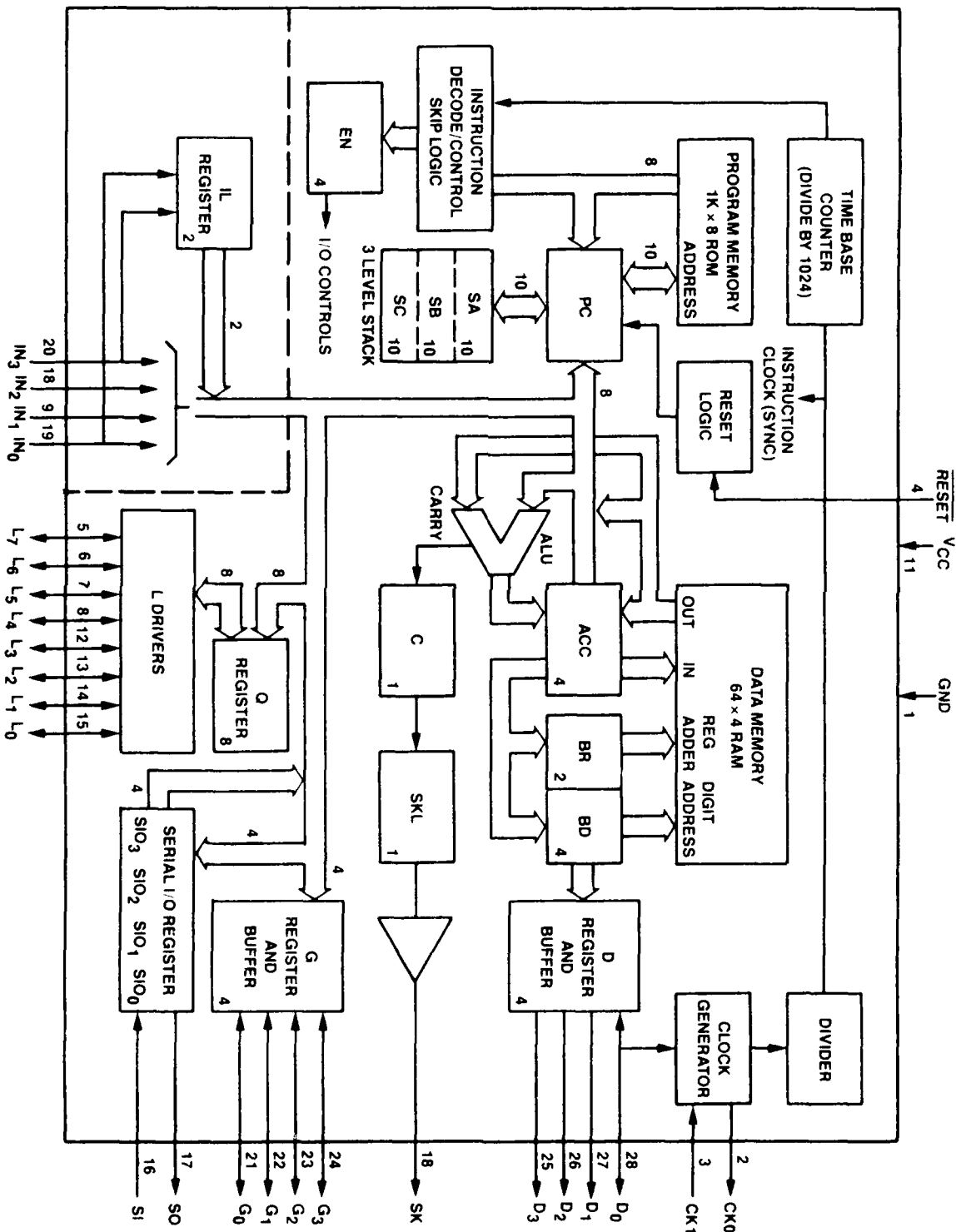


Figure C-1. Microcomputer Block Diagram.

(6) The Q-register is an internal, latched, 8-bit register, used to hold data loaded to or from ROM and the ACC-register. Its contents are output to the L I/O ports when the L-drivers are enabled under program control.

(7) The contents of the eight L-drivers may also be read directly into ROM and the ACC-register.

(8) The SIO-register is used as a serial input register to read several data samples on the SI input port.

(9) The L-ports and the SIO-register are controlled by the EN-register. When EN = 5, the data in the Q-register is latched to the L-drivers.

(10) The 10-bit time base counter divides the instruction cycle frequency by 1024 providing an overflow condition. This feature generates the time base for providing a real time count. This counter is used to monitor the internal timing of the critical functions provided by the microcomputer.

2. Microcomputer Software.

a. The fuze software may be broken into five major areas: program initiation, arm/delay switch sampling and decode, retard verification, arming sequence, and fire delay programming.

(1) An external reset pulse is provided at power-up to keep the minimum initialization time for the microcomputer fixed. Upon initialization, the program counter (PC) and several other registers internal to the microcomputer are cleared to zero. The flow chart for program initiation is given in Figure C-2. The portion of RAM used during program initialization is tested and cleared prior to storage of data in RAM. The first decision block (diamond) in the flow chart checks the turbine release signal to select between the turbine (Air Force) mode or the FFCS (Navy) mode. In the turbine mode, flags M(3,9,0) and M(3,9,2) are left in the zero state and the arm time is initiated by the execution of the first instruction. For the FFCS mode the flags M(3,9,0) and M(3,9,2) are set to "1", and a loop is entered to detect when the FFCS signal has released and to synchronize this time with the internal program timers. After exiting the FFCS release loop the magnitude and polarity data are strobed and stored in memory, for arm and delay times, prior to rejoining the turbine software flow. The arm and gag switches are then sampled to insure the fuze has not been armed or ungagged. The microcomputer will dud the fuze if the switches indicate ungagged or armed. The RAM locations used during the remainder of the program are now tested and cleared using subroutine "RAMTST". This RAM test will dud the fuze if it detects a RAM bit failure.

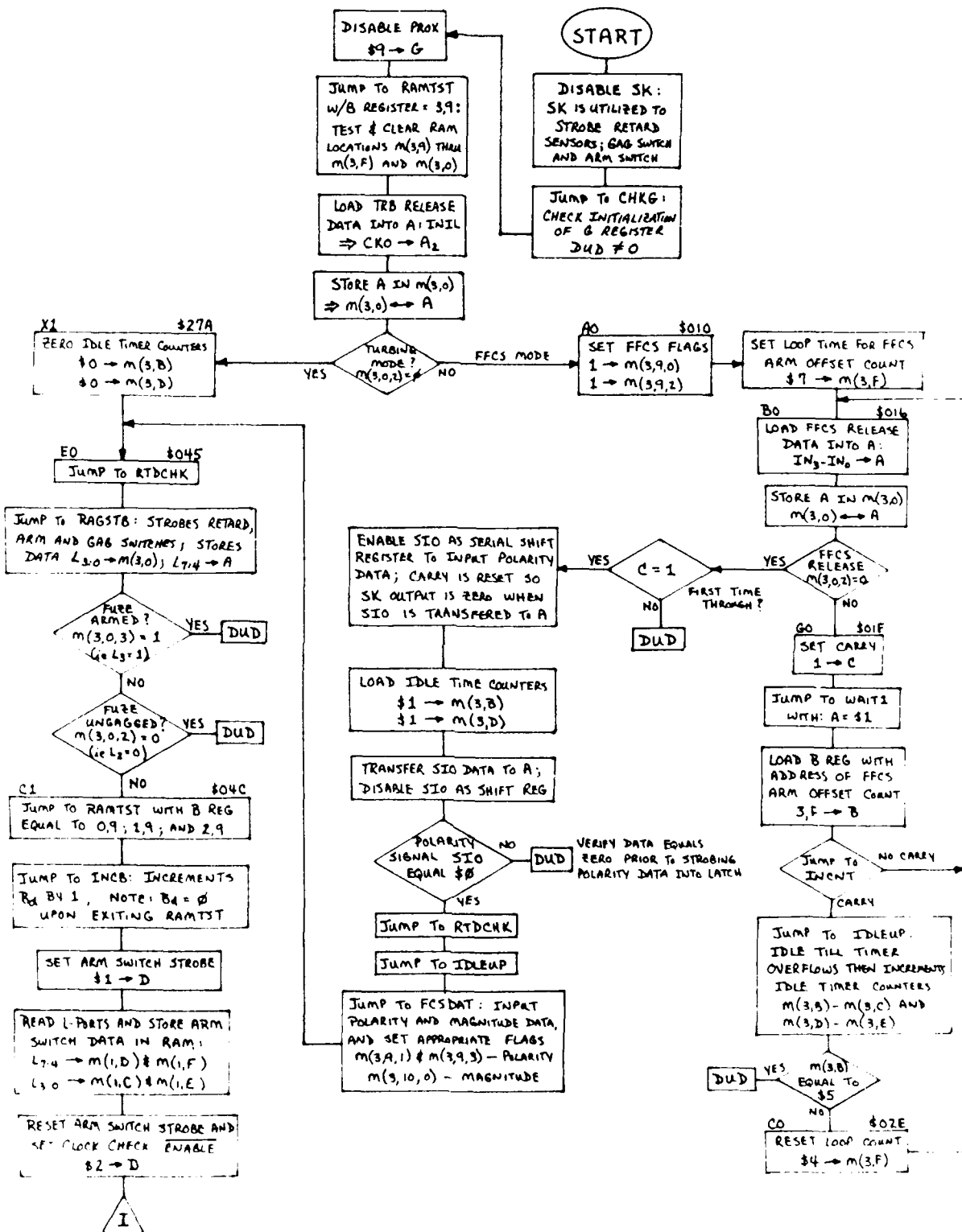
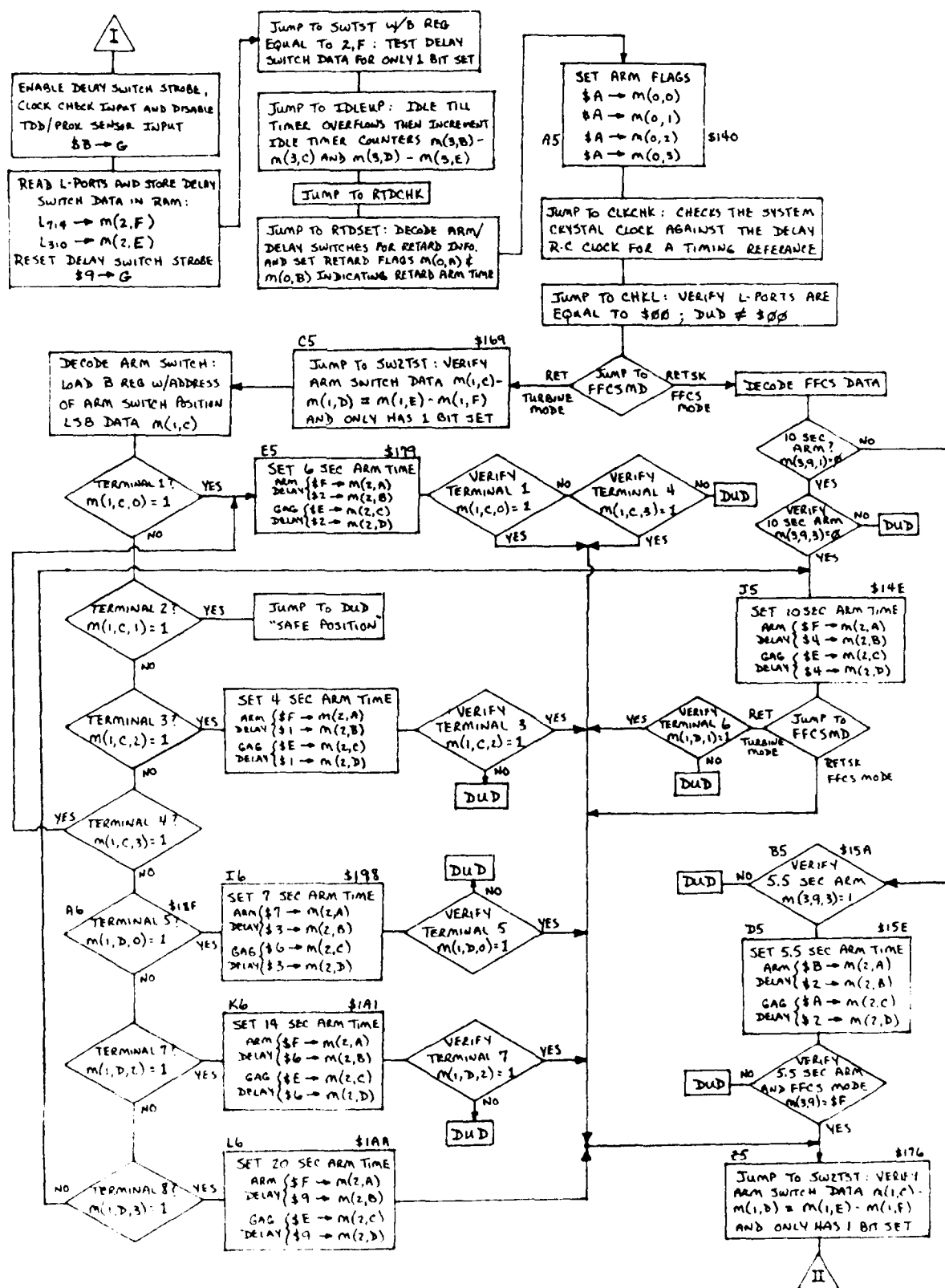


Figure C-2. Program Initiation Software Flow Chart.



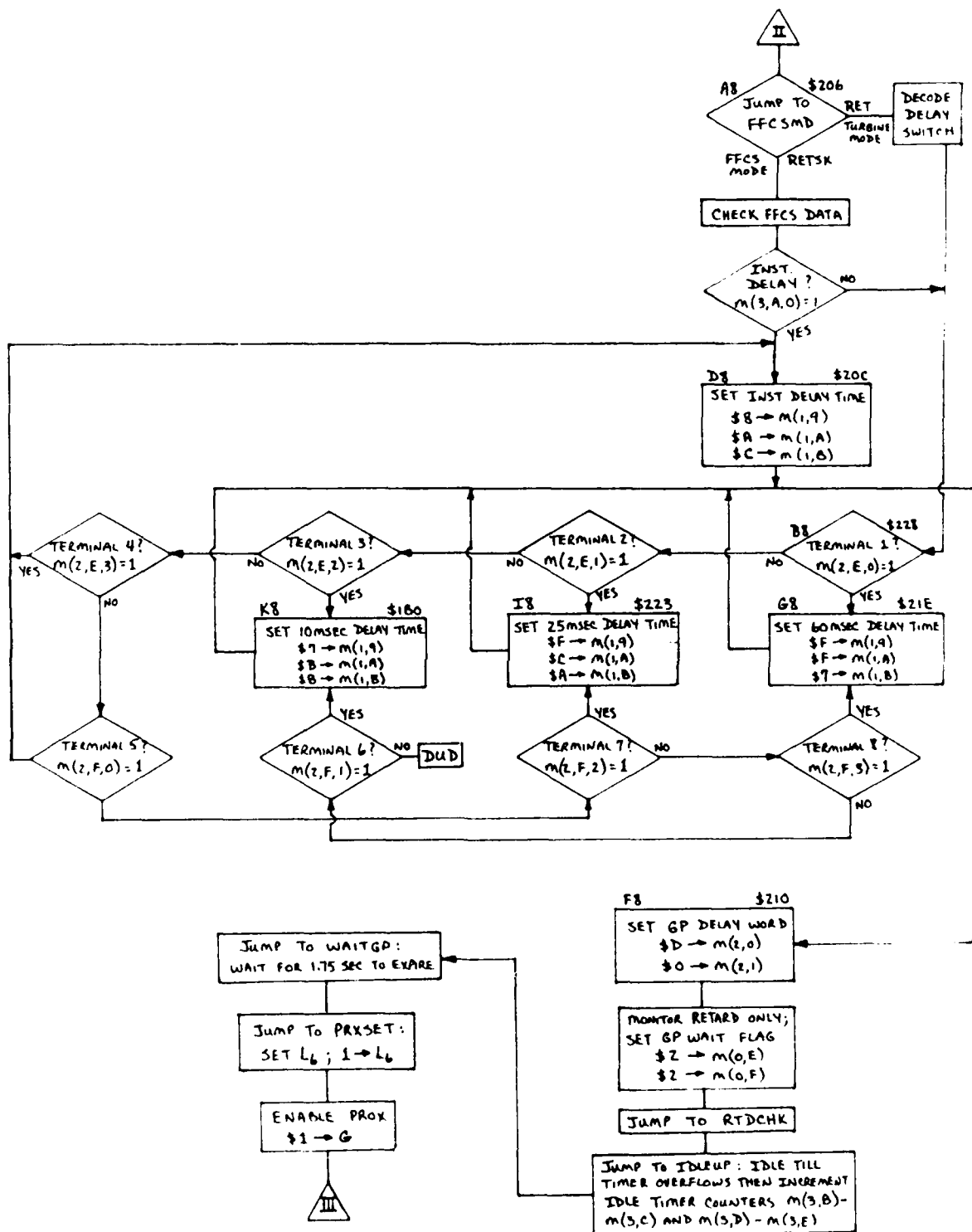


Figure C-4. Delay Switch Decode Software Flow Chart.

(2) The arm/delay switch sampling is given in Figures C-3 and C-4. The arm switch decoding is given in Figure C-3 and the delay switch decoding is given in Figure C-4 of the flow chart. The arm and delay switches are strobed and stored in memory. The arm switch data is stored in redundant memory locations M(1,12/13) and M(1,14/15), and the delay switch data is stored in memory location M(2,14/15). Each switch is tested to insure only one switch position is decoded. The microcomputer will dud the fuze if more than one bit, in the switch data words, is set indicating a switch failure. The subroutine "RTDSET" is now executed to flag, in redundant memory locations M(0,10) and M(0,11), the selected retarded arm time. The four arm flags M(0,0) through M(0,3) are set prior to decoding the switch data. These flags are set to a "1010" binary pattern and are checked prior to arming to insure this portion of memory was executed. The next decision block divides the turbine and FFCS flow for decoding and setting of the arm/gag delay times (Figures C-3). In the turbine mode, the redundant arm switch data memory locations are checked for equality and the microcomputer will dud the fuze if they are not equal. The arm switch data in M(1,12/13) is decoded and the proper arm/gag delay times are stored in memory. After the delay times are stored in memory, the arm switch data is verified to insure the proper delay times were set. The FFCS arm/gag delay times are set and verified from the magnitude and polarity data in memory. The delay after impact times are then decoded (Figure C-4) and set in memory.

(3) At this point, the software enters a general purpose wait (WAITGP) subroutine and waits for 1.75 seconds to expire while monitoring the retard switches. Through this portion of software, the retard sensors have been sampled twice every 125 ms. Upon receiving a valid retard sample, a set of redundant memory (M(0,12) and M(0,13)) locations are incremented and tested for 16 and/or 14 counts. Memory flag locations M(1,0,1) and M(1,0,3) are set when 14 counts have been received, and M(1,0,0) and M(1,0,2) are set when 16 counts have been received. There are three general paths (Figure C-5) for the retard verification software depending on the selected retard time. Retard arm flags M(0,10) and M(0,11) determine which path is taken. In each path, memory location M(1,0) is checked for the proper number of retard counts. If M(1,0) is equal to "1111" (hexidecimal "F") indicating the microcomputer has received 16 samples of retard switch closures, the software will set the selected retard arm time in memory. After the selected retard arm time is set in memory, the retard arm flag and retard count are verified before proceeding in software. If M(1,0) is not equal to "1111", the software will jump to the "WAITGP" subroutine until the selected gag delay time has expired. In the "WAITGP" subroutine, the delay word is compared to the Idle Timer counters to insure the delay time has expired prior to exiting the subroutine.

(4) The arming sequence is initialized when the PAF capacitor is charged for firing of the piston actuator (Figure

C-6). This occurs 125 ms prior to the selected arm time. The piston actuator is fired 100 ms prior to the selected arm time. After UNGAG fire, the residual energy on the PAF capacitor is removed in subroutine "DUMP". The PAF capacitor is charged for arming 30 ms prior to the selected arm time. The gag shear wire switch is then checked to verify if the switch is open. Arm flag memory locations M(0,0) through M(0,3) are then checked for the appropriate value before proceeding with the arming sequence. The arm delay time is then checked against the Idle Timer counters to insure the appropriate time has elapsed. The software then makes one final test of the impact and proximity sensors before the arm pulse is generated. After the arm pulse is generated, the software verifies the closure of the arm switch before proceeding with the delay programming for the fire signal.

(5) After verification of fuze arming, the microcomputer transfers its remaining task to external circuitry. The microcomputer programs an external shift register to the number of pulses stored in M(1,9/11) (Figure C-6). These counts are required to realize the various impact delay times. After this count is stored in the shift register, the delay clock circuit is enabled. This allows the input of an impact or proximity signal to initiate the fire delay signal. The software then enters a loop which replenishes the charge on the PAF capacitor every 10 seconds. This loop is continued until a fire signal is received.

b. The flow charts for the various subroutines are not included in this report. Two of the more important subroutines which were not discussed in the above description are the "DUD" and "CLKCHK" subroutines.

(1) The "DUD" subroutine is used when the software detects a failure to dud the fuze. It enables the PAF charge switch and a short across the PAF capacitor simultaneously to deplete the energy stored in the primary energy storage capacitor and the PAF capacitor.

(2) The "CLKCHK" subroutine utilizes the delay clock to monitor the frequency of the crystal clock. This subroutine enables the delay clock to clock an external 12-bit binary counter which is compared to the accumulator carry of the microcomputer. The accumulator carry is dependent on the crystal oscillator frequency and the binary counter output is dependent on the delay clock frequency. This subroutine is used throughout the flow diagram to dud the fuze if the crystal oscillator frequency is out of tolerance.

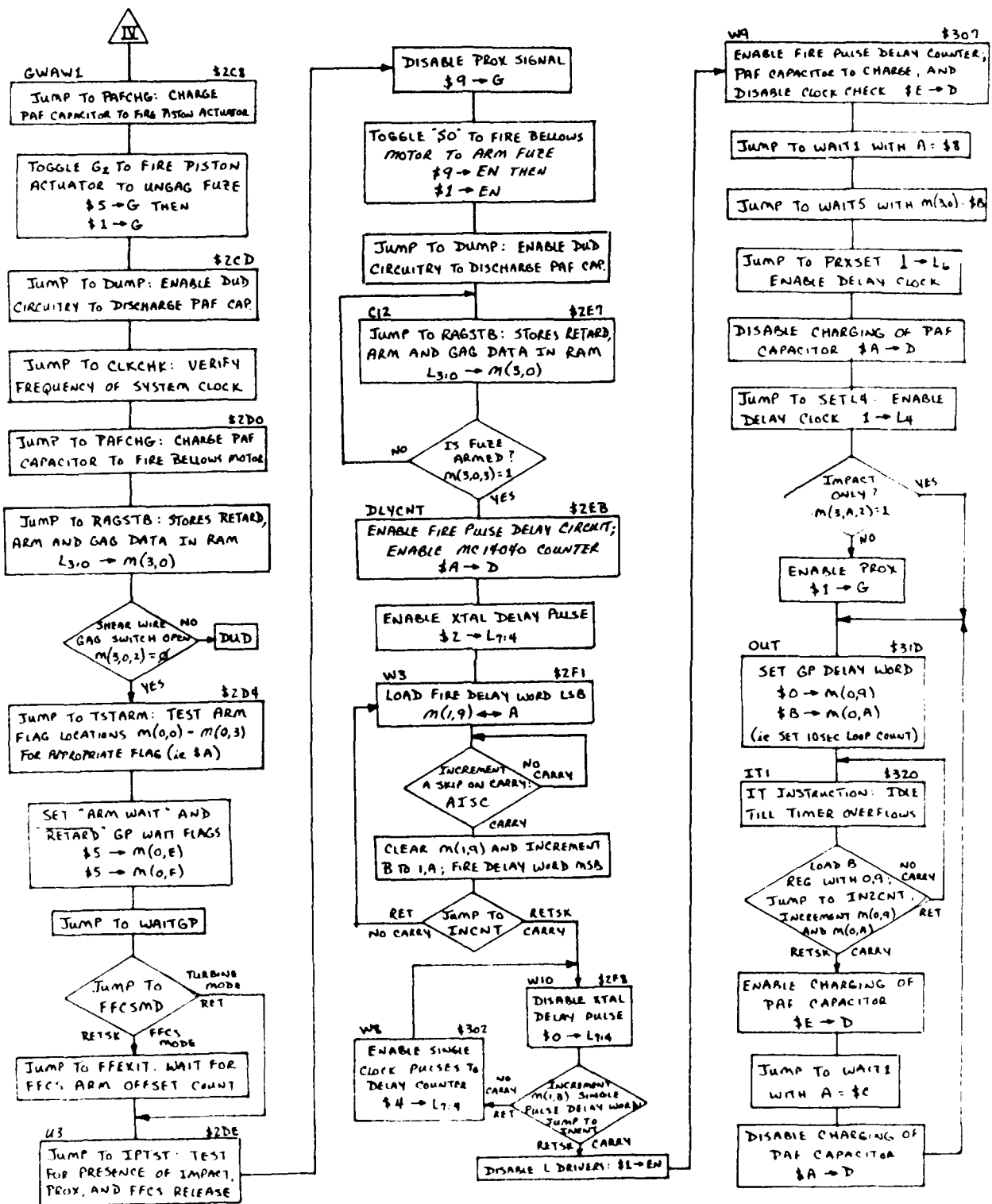


Figure C-6. Arming Sequence and Fire Delay Programming Software Flow Chart.

DTIC

END

4-86